ESD-TR-75-54

MTR-2932, Vol. 1

A SOFTWARE VALIDATION TECHNIQUE FOR CERTIFICATION: THE METHODOLOGY

D. E. Bell
E. L. Burke

April 1975

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 522B
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-75-C-0001

REVIEW AND APPROVAL

"This technical report has been reviewed and is approved for publication."

[illegible]
WILLIAM R. PRICE, 1Lt, USAF
Project Engineer

[illegible]
MARVIN E. BROOKING
Project Officer

FOR THE COMMANDER

[illegible]
ROBERT W. O'KEEFE, Colonel, USAF
Director, Information Systems
Technology Applications Office
Deputy for Command & Management Systems

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-75-54	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A SOFTWARE VALIDATION TECHNIQUE FOR CERTIFICATION: THE METHODOLOGY		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER MTR-2932, Vol. 1
7. AUTHOR(s) D. E. Bell, E. L. Burke		8. CONTRACT OR GRANT NUMBER(s) F19628-75-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation, Box 208, Bedford, MA, 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522B
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems, Electronic Systems Division, AFSC, Hanscom Air Force Base, MA, 01731		12. REPORT DATE APRIL 1975
		13. NUMBER OF PAGES 37
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Security
Certification
Security Kernel
Proof-of-Correctness
Software Validation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Certification is the approval, by some appropriate authority, that a system meets some functional criteria. In the past, critical software systems, such as security controls have not been certifiable because of the unavailability of a formal validation technique. This paper establishes such a formal methodology for validating the correctness of a software system. The methodology is both rigorous and general and is suitable for certifying the effectiveness of software security controls that are to be used in an open environment. A companion volume will develop a detailed example based on a security

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract (Continued)

kernel for a PDP-11/45.

FOREWORD

For the past two years, The MITRE Corporation has been pursuing a solution to the problem of controlling security in computer systems. This effort, sponsored by the Electronic Systems Division of the Air Force Systems Command, specifically addressed the military security scheme. Because many procedures are available in the military for communications, physical, and personnel security, this effort has concentrated on the design and development of provably effective hardware and software access controls. This document addresses the problem of utilizing theoretical results about computer security in an actual design and implementation task.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

SECTION I
INTRODUCTION

BACKGROUND

The certification of software systems has not, in the past, been firmly rooted in sound engineering techniques. During the development of a software security control system, or a security kernel, it became clear that if the kernel was to be used in an environment open to the malicious user, a very strong guarantee of the kernel's ability to protect against information compromise would be necessary. The available techniques for certification were scrutinized and found to rely heavily on ad hoc examination and approval of software. Software engineering techniques, such as structured programming, proof-of-correctness and automated aids for program proving, were available, but a comprehensive plan for using these techniques in a cohesive software validation effort did not exist. This paper develops such a cohesive software validation technique that is applicable to the certification of critical software systems. This certification effort is part of a broad program in computer security outlined by Anderson [1].

CERTIFICATION

The term "certification" refers to the approval, by some designated authority, of some software system. This software is then said to be certified to perform some function. Taking the case of a security kernel as an example, the functionality that must be certified is absence of the possibility for compromise. The judgement of certifiability is based on both the design of the security kernel and on the risks perceived to be in the computer's environment. A computer system operating in a totally benign, physically protected environment may not need to have its hardware and software access controls certified against a malicious user. In the more general case of an open environment, however, reference-monitor-based security controls provide the protection against compromise of classified information. Because of the nature of the environment only the security kernel stands between an uncleared individual and classified information stored in the computer system. This security kernel must, therefore, undergo a great deal of scrutiny before it can be certified.

The technique developed in this paper provides the necessary scrutiny for software security controls that are to be used in an open environment. Because the technique can be used wherever a mathematical model is available, the technique is quite general.

Software validation is an engineering technique used to prove things about the behavior of software. In the context of computer security, validation is meant to be sufficient for certification of a security kernel in any environment. The goal of validation is dual; it is both to demonstrate that a proposed solution <u>is</u> a solution as well as to make the demonstration itself convincing to those who have the authority to approve a system for actual use.

Hence, the process of validation must provide a sound and rigorous justification for a proposed solution as well as full and open documentation of the validation effort designed to win the confidence of any disinterested party.

OVERVIEW

This paper establishes forms for all representations of the system design from the most abstract mathematical model to the realization of this model in binary machine language on some hardware base. The paper then goes on to show how each representation can be proven to correspond to the more abstract representation that precedes it.

The methodology developed in this paper was developed for the certification of a security kernel. Because many of the concrete examples are peculiar to security control, the next section, Section II, reviews the overall approach being taken in the security development effort. Section III discusses the various representations of the software system and the relationships between these representations. Section IV deals with proving the correspondences between all representations. Section V summarizes the key ideas of the paper. A companion volume will provide an explicit example taken from the certification of a PDP-11/45 security kernel.

SECTION II

COMPUTER SECURITY CONCEPTS

INTRODUCTION

Because the software validation effort was motivated by the stringent requirements for correctness imposed by the computer security program, a brief discussion of the important aspects of that program is presented below. The history and direction of the computer security program are outlined by ESD [2].

REFERENCE MONITOR

The ESD/MITRE computer security program centers around the

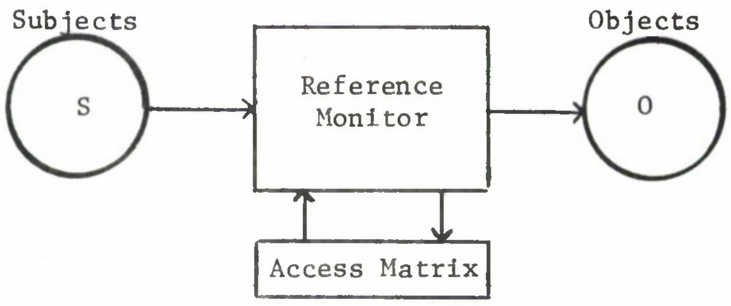

Figure 1. Reference Monitor

concept of the reference monitor. The reference monitor controls access to objects (files of information) by subjects (people, or the processes that operate on behalf of people) and has three characteristics that insure that it provides security:

1. it mediates all access attempts according to the rules of the DoD Security System;

2. it is protected (usually through isolation) from the remainder of the software;

3. it is provably correct.

The reference monitor is realized in the hardware and software mechanisms needed to implement this concept on a computer, and the software portion of the monitor is called the security kernel.

MATHEMATICAL MODELS

In order to describe explicitly how the reference monitor works, mathematical models have been developed. Two models will be discussed. One model, developed by Bell and La Padula [3-5] is based on general systems theory, specifically, dynamical systems theory. The abstract reference monitor is represented in this model as rules that govern changes of state. A state is the aggregate of several variable quantities - the current-access set, the access matrix, and the classification functions. The concept of security is included in the model with the definition of a secure state. The principal result of this model is the rigorous proof that the state transitions allowed by the rules of the reference monitor prohibit the system from reaching a compromise state (that is, a non-secure state).

A second mathematical model was developed by Walter *et al* [6]. This model represents the reference monitor in the most abstract sense and attempts to use the technique of function decomposition to arrive at a mathematical model that ultimately can guide an implementation.

The functional decomposition approach identifies access functions in the most abstract model. Subsequent mathematical models refine these functions into their constituent parts until the access control is defined by compositions of functions. The functions are refined until they correspond to the desired level of detail, i.e., until they are specific enough to directly guide the implementation. The use of this model to synthesize a software system will also be investigated.

These two models are by no means the only formal mathematical models of secure systems. Popek [7] and Hsiao _et al_ [8] are two other examples of abstract models of secure systems. In each case, there is the notion that some formal technique must guide the eventual design, because informal techniques are inadequate.

IMPLEMENTING A SECURE SYSTEM

The implementation of a secure computer system clearly requires careful planning and analysis. Our analysis led to the development of an abstract model for the reference monitor. The process of model development gave the participants a certain insight into security-related problems as well as specific implementation guidelines for topics directly addressed in the model. However, the necessity of absolute algorithmic security in the final implementation of our system made it obvious that neither of these benefits was sufficient to complete the task of implementing a secure system.

The translation from the model to a useable computer system must be done just as carefully as was the development of the model. The criteria for the design scheme is that the behavior of the software on the machine must, in some appropriate sense, be equivalent to the behavior of the mathematical model. The remainder of this paper is devoted to a discussion of how the design should proceed in order to guarantee the ultimate validation of the system with respect to the mathematical model.

SECTION III

COMPONENTS OF A SOFTWARE DESIGN

INTRODUCTION

In this section, we will advance and discuss a set of components for software design. In the detailed discussion later in this section, we will list the purpose and nature of each of the components. Before we treat the components individually, however, we should explain the framework for system development that we are advocating.

The process of validation, mentioned briefly in Section I, has as its goal the clear and rigorous proof that a conceptual solution to a real-world problem has been precisely implemented on a particular hardware/software "machine" that is to deal with that real-world problem. To simplify this task, we propose the use of the validation chain shown in Figure 2.

The use of the validation chain allows a solution to be evaluated in several small steps rather than in one massive leap from the "machine" to the real-world problem.* Moreover, the subdivision of the problem makes possible the validation of several particular solutions using several common blocks in the validation chain. For

*The full development of a mathematical model, of course, will normally imply agreement among experts that the statement of the problem in the model accurately reflects the real-world problem. Hence at this point the problem is reduced to demonstrating an appropriate correspondence between the model solution and the "machine" solution.

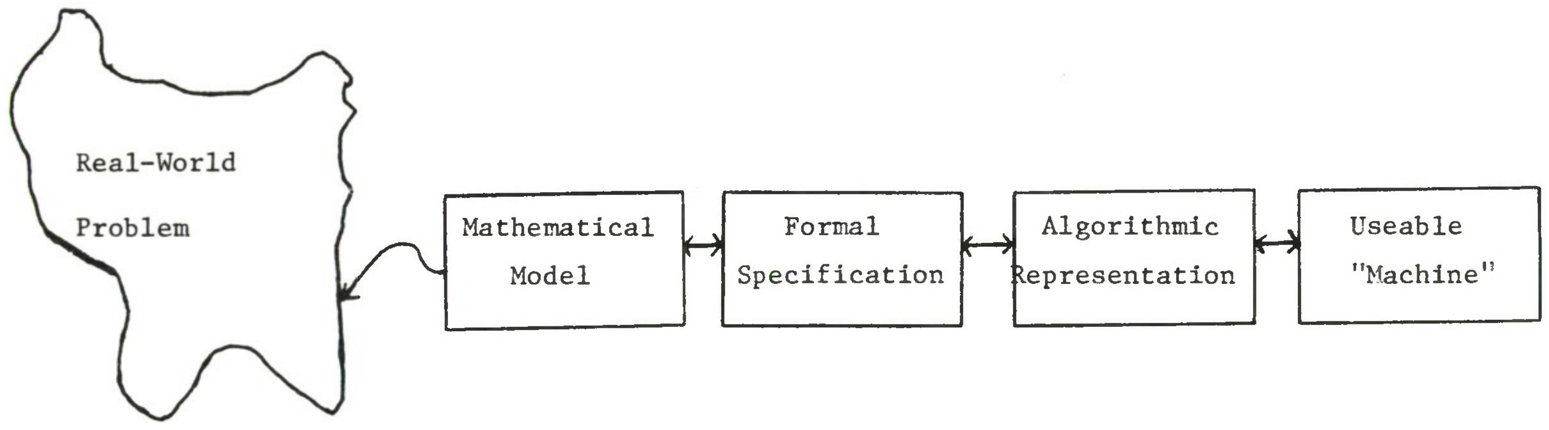

Figure 2. The Validation Chain

example, if one model is made of a problem and one formal specification corresponding to the model is developed, then two particular solutions for two different computers could be validated by completing the validation chain in two different ways. Similar savings of efforts could be realized at any stage of the chain, even by compiling the source code differently, necessitating only one new validation link.

The use of the validation chain will require a clear understanding of the purpose and nature of the constituent blocks and of the links between them. The first of these topics will be discussed in the remainder of this section, and the second is the topic of Section IV.

SYSTEM COMPONENTS

Each of the blocks in the validation chain represents a solution to the problem at a different level of detail. The Mathematical Model addresses a pure abstraction of the problem. The purpose of the model is to describe and then to solve the problem conceptually. The Formal Specification provides a blueprint for the organization and structure of acceptable software implementations of the model's solution to the problem. The Algorithmic Representation is one particular instance of an implementation blocked out by the Formal Specification. The Useable "Machine" is the combination of a particular computer operating with the object code generated by a particular compiler from the higher level source code that is associated with the Algorithmic Representation. The "Machine" thus depends not only on the conceptual design (at the level of detail specified by the Algorithmic Representation) but also on the semantics implicit in the compiler and on the computer itself.

As might be expected, the natures of the various components differ. Let us now discuss each of the components in turn.

Mathematical Model

The starting point of the software system design is the mathematical model, the leftmost block of the validation chain. The mathematical model is an abstract representation of the variables of the system and of the operations allowed on these variables. For security control, the mathematical model describes abstractly the behavior of the reference monitor.

The Mathematical Model will normally be expressed in abstract mathematics. General systems theory, differential equations, automata theory, abstract algebra and linear programming are some likely branches of mathematics that might reasonably be expected to be used in the development of a mathematical model. In the consideration of computer security, very simple, nonstructured constructs are the most useful. In the Bell-La Padula model [3-5], general systems theory is used, while the CWRU model [6] uses basic set theory.

The correspondence of the real-world problem to the problem statement in the model must be agreed upon by competent experts in the field. The correctness of the model solution to the problem is established by proving theorems based on the model's definitions and its statement of the problem. When the ability of the model to describe the problem is accepted and when a correct solution has been formulated and verified, the model becomes the standard for the other blocks in the chain, in the manner to be discussed in Section IV.

Formal Specification

The mathematical model deals with abstract entities that must be realized in a concrete fashion. The first step in the process of realizing the model abstractions is to impose restrictions on the abstract entities of the model and express the resulting system as a formal specification. This specification completely identifies the state variables of the representation and all the functions that a user might invoke to observe or modify one of these state variables. As an example of the constraints placed on the model in the formal specification, consider the security model of Bell and La Padula [3-5]. This model deals with abstract entities called objects. In the realization of any system based on the model, such as that by Schiller [9], the objects must be given certain attributes like type and size. Object type and size then become state variables and functions must be provided in the formal specification to observe and manipulate these variables.

A possible format for the formal specification is that developed by Parnas [10]. Parnas specifications have been used successfully by Price [11] and Schiller [9] and will be used by Neumann, *et al* [12] in efforts where statements about the behavior of the system must be proved. It may be possible to circumvent the formal specification by choosing a sufficiently rich language for the algorithmic representation. Such a language will be considered below.

Algorithmic Representation

The functions of the formal specification must eventually be realized by a set of algorithms, or programs. Thus the next representation of the software system is in terms of algorithms, or program modules. One of the motivations for the formal specification (according to Parnas) is to hide the details of the implementation so that design decisions are not made solely to expedite the implementation. Since the formal specification has decomposed the system into a series of function modules, it should be a fairly straightforward problem to implement each of the modules in some suitable high-level implementation language.

The programs must then be proven correct with respect to a series of assertions, and these assertions are derivable from the formal specification and the correspondence mappings. The relationship of this proof technique to work in proof-of-correctness will be examined shortly.

It was stated previously that it may be possible to eliminate the formal specification. Since a goal of this software synthesis technique is to provide a methodology for realizing abstractions, then if an algorithmic language were to exist that had a sufficiently rich structure for expressing the abstractions of the model and for refining these abstractions within the language, then this language could serve as both the formal specification and the algorithmic representation. The work of demonstrating a correspondence between the two representations could then be done in the development of the programs themselves. Such a language, called CLU for the abstract data clusters it supports, is under development by Liskov and Zilles [13]. This language has been used by Karger [14] in an attempt to express the model of Walter, et al. [6]. When CLU becomes more completely specified, its utility in such a software engineering approach should become established.

Useable "Machine"

Eventually a program is translated from the high-level language representation into binary machine language and is run on a particular computer. Although this component (the hardware/software machine) is often ignored in the literature of software reliability, its behavior is by far the most important. The notion of correctness is perhaps fuzziest when applied to the machine language representation. While specific guidance is given the formal specification and algorithms from the model and specification, respectively, little guidance on "correctness" comes to the machine language representation from the algorithmic representation. The model's theorems dictate the functionality of the formal specification and relations true about the formal specification dictate the assertions about the algorithmic representation. Because high-level languages usually lack a formal defintion of their semantics, it is difficult to make formal inferences about the machine language code. The correctness of the machine language version will rest with establishing a semantics for the machine language used and verifying the correct interpretation of this semantics by the hardware.

A PREVIEW OF CORRESPONDENCE METHODOLOGY

Now that each component of the software system has been described, it would be useful to preview the rest of the methodology; in Section IV we will show the relation of this methodology to previous work in software reliability and proof-of-correctness.

The remainder of the software synthesis technique may be clear at this time. Given the four representations described previously, it remains only to show how one goes about proving the correspondence between each pair of successive representations. Figure 3 illustrates how the correspondence will be demonstrated between any two consecutive representations. The methodology will start with a mathematical model whose

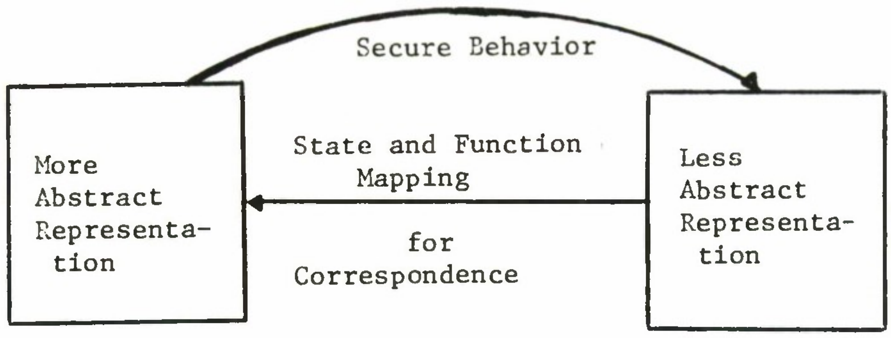
Secure Behavior
More Abstract Representation
State and Function Mapping
for Correspondence
Less Abstract Representation

Figure 3. Proving Behavioral Correspondence

behavior is proven secure. The object is then to prove that the behavior of each subsequent representation corresponds to the (proven secure) behavior of the previous representation. The proof is done, as shown in Figure 3, by mapping the states and functions of the less abstract representation into the states of the previous more abstract representation. A state of a representation is determined by the values of the state variables and a state mapping identifies corresponding states in each representation. The functions that change the state of the more abstract representation have been proven acceptable in the sense that they have been shown to always change one secure state to another secure one. By showing that the functions of the less abstract representation correspond to the functions of the more abstract representation, the behavior of the less abstract representation is shown to be secure. The next section develops these notions of correspondence in a formal way and describes the characteristics of the proofs.

SECTION IV

CORRESPONDENCE PROOFS

INTRODUCTION

At the heart of the validation technique what we are proposing is the notion of correspondence: the validity of the solution that a final hardware/software machine provides for a real-world problem is assured by its "correspondence" to the solution of the abstract problem addressed in the model. It is the purpose of this section to discuss correspondence proofs by addressing, in turn, the general nature of correspondence, a precise mathematical description of correspondence and techniques for use in the various correspondence proofs along the validation chain.

THE NATURE OF CORRESPONDENCE

The "correspondence" that is required between blocks of the validation chain is at first glance different from a proof of program's correctness. However, a substantial case can be made for essential identity of the two concepts [15]. In particular, what is required in each case in some sort of demonstration that certain transformations in one area of discourse correspond in some satisfying way to expected transformations in another area of discourse. This demonstration must involve a type of mathematical proof, whether expressed mathematically or not. This type of proof, as a mathematical process, cannot be generally specified although helpful techniques can be listed [16-18]. However, the form that the proof must take can be elucidated using a branch of mathematics called "category theory".

Category theory can be used to structure discussions of correspondence proofs through its ability to address representations of finite-state machines. Since each of the blocks of the validation can be conceptualized as a finite-state machine, the proof of a correspondence between adjacent blocks of the chain can be phrased as a demonstration of a kind of relation between finite-state machines. In particular, it is necessary to show that each block is represented by the block to its immediate left. Thus, any state transformation within the right block of a link in the validation chain must be shown to correspond to an allowable state transformation in the left block. Hence, the relation of category theory to correspondence proofs can be addressed at the level of abstract automata, as will be explained in the next subsection.

A MATHEMATICAL SPECIFICATION OF CORRESPONDENCE

Let $A = (X, Y, K, \delta, \lambda)$ be an abstract automaton [19], where X is the set of inputs; Y, the set of outputs; K, the set of states; and δ and λ, the state-transition and output functions, respectively. For every pair of states k_1 and k_2, there are potentially many input strings I such that $k_2 = \delta(k_1, I)$. Thus, the set of transformations $k_1 \longrightarrow k_2$ can be related to sets of input strings I. In particular, a transformation α: $k_1 \longrightarrow k_2$ will be associated with k_1 and an input string I such that $k_2 = \delta(k_1, I)$. The set K of states together with all state transformations associated with input strings make up a mathematical structure known as a category.

A category $C = (\mathcal{O}, \mathcal{M})$ is a set $\mathcal{O}$ of objects together with a set $\mathcal{M}$ of morphisms.* A morphism can be thought of as a transformation from an object O_1 to another object O_2. For this reason, the set $\mathcal{M}$ is frequently thought of as the disjoint union of morphisms from object O_1 to object O_2, as O_1 and O_2 range over $\mathcal{O}$:

$$\mathcal{M} = \bigcup_{O_1, O_2 \in \mathcal{O}} \hom(O_1, O_2)$$

*Basic notions of category theory can be found in [20].

The morphisms of $\mathcal{M}$ must have an associative composition and there must be an identity morphism for each $0 \in \mathcal{O}$.

Expressing automata in category theory is very straightforward. For the automaton $A = (X, Y, K, \delta, \lambda)$, we define the associated <u>state category</u> $C_A = (\mathcal{O}_A, \mathcal{M}_A)$ as follows:

$\mathcal{O}_A$ is the set K of states;
$\hom(k_1,k_2)$ is the set of pairs (k_1,I) where I is an input string such that
$k_2 = \delta(k_1,I)$ for states k_1 and k_2; and

$$\mathcal{M}_A = \bigcup_{k_1,k_2 \in K} \hom(k_1,k_2).$$

The demonstration that C_A is a category is direct and is thus omitted.

The problem of proving correspondence involves two automata, the test automaton A and the specification automaton B (the right and left blocks of a validation link, respectively). The purpose is to demonstrate that A is represented by B so that statements about B are applicable to A. The importance of category theory is that, in some limited sense, one can compare the categories of apples and oranges (contrary to popular wisdom) by establishing a functor between them.

In category theory, a functor relates one category to another category while preserving the composition of morphisms. More formally, if C and C' are categories, then F is a functor between C and C' provided:

1. for every object 0 of $\mathcal{O}$, F specifies an object 0' of $\mathcal{O}'$;
2. for every morphism $\alpha \in \hom(0_1, 0_2)$ in C, F specifies a morphism $F(\alpha) \in \hom(F(0_1), F(0_2))$; and
3. the diagram below commutes (that is, the action of α followed by translation into C' is the same as translation into C' followed by the action of $F(\alpha)$:

$$\begin{tikzcd} 0_1 \arrow[r, "\alpha"] \arrow[d, "F"'] & 0_2 \arrow[d, "F"] \\ 0_1' \arrow[r, "F(\alpha)"'] & 0_2' \end{tikzcd}$$

A correspondence is a functor from the state category of the test automaton A to that of the specification automaton B. The demonstration of a correspondence would show:

1. the explicit interpretation of a state in A as a state in B;
2. the explicit interpretation of a state transformation in A as a state transformation in B; and
3. that under interpretation the action of A "corresponds" to the action of B (in the sense of the commutativity of the diagram above).

The correspondence that is chosen in the course of using the validation must be carefully chosen and no specific guidance for its choice can be given. The appropriateness of the choice will be predicated on the ability of the people involved and established by critical review of their work by the widest possible community.

Through our use of the validation chain, we intended to divide the validation task into several smaller tasks. The specification of correspondence in categorical terms not only makes explicit the requirements of a proof of a correspondence, but also provides a theoretical basis for the subdivision of the tasks. In particular, since the composition of functors is a functor, successful demonstrations of correspondence at each link of the chain guarantees that the final solution "corresponds", in the rigorous sense, to the conceptual solution of the leftmost block of the chain. Moreover, the development of specific validation techniques is now structured by the goal of proving correspondences that has been carefully delineated.

TECHNIQUES FOR PROVING CORRESPONDENCES

General

In the framework of the validation chain that we are proposing, the demonstration of a functor between the state categories of adjacent blocks will vary from link to link. This variation stems primarily from the differing modes of expression used in the various blocks. In general, however, the constituent activities will have a common flavor. In the second volume of this report, the various steps of the technique will be illustrated; in the remainder of this subsection, the basic parts of the process will be discussed in general.

First, from the overall description of each representation there must be extracted a full list of the states and of the possible state transformations. The specification of a state will rely on a full list of state variables and any restriction on combinations of variables.

Next, a "translating dictionary" between the states of the blocks in the link must be constructed for the link. Where the state is a vector of state variables, an association between the variables of one representation and the variables of the other representation would be sufficient. In fact, this arrangement is conceptually the simplest. If certain combinations of variables are to be associated, the situation would be made more complex, but this course cannot be ruled out as unncessary in all circumstances. Such a translating dictionary is the practical analog of the object map of the state category functor.

The next step is the analog of the morphism map of the state category functor. It involves specifying for each state transformation in the test automaton A a corresponding state transformation in the specification automaton B. By the associativity of morphism composition, it suffices to consider only input symbols in A, since any input tape I in A is the concatenation of such symbols. Hence, the task degenerates to establishing an input tape in B to correspond to each input symbol in A. This assignment must be done with an eye towards the last step in the process, the commutativity check.

The demonstration of the commutativity of the functor diagram is a check that for every state v_1 of automaton A and every irreducible transformation α to state v_2, the translation of v_1 to automaton B transformed by the transformation corresponding to α yields the same state as the translation of v_2 to B. Clearly, this process will be straightforward, even if somewhat tedious. A failure to arrive at the same state would be stimulus to try one of several courses of action.

1. Check the demonstration itself for errors.
2. Check the object and morphism maps. With the knowledge of what error turned up, it may be possible to alter the morphism-map image of α to allow completion of the step.

3. If no reinterpretation of A can rectify the situation, the information gathered in this effort should suggest the changes to A which should be made to make A correspond to B.

The Model-to-Specification Correspondence

The model is likely to be written in terms of sets and functions while the abstract specification will probably be in a Parnas-like specification language [10]. The differences between these modes of expression will probably prove the greatest hurdle in proving this correspondence. The task of establishing a translating dictionary could be formidable here. There are no special techniques that would appear to aid in this endeavor. This correspondence will involve grinding through the details of a full-fledged proof.

The Specification-to-Algorithm Correspondence

Both of these representations are likely to be phrased in formal languages. There is, furthermore, the likelihood that state variables and state transformations will have an almost transparent correspondence in the use of similar names. Hence, the problem here is essentially one of translation from one language to another. In addition, if the correspondence of specifications to progam modules is direct and simple, the situation is precisely that of traditional proof-of-correctness, with the benefit of an explicit standard against which to "prove" the program.

The Algorithm-to-"Machine" Correspondence

This correspondence seems to pose the most practical problems. In particular, the general problem here presumes an understanding of the semantics of the higher-level language used in the algorithmic representation. Unfortunately, formal semantics is in its practical infancy and little useful work in this area has been done.

The general problem of proving that a machine-language translation corresponds to a higher-level program admits of two general approaches.

The first approach is to establish that the compiler used generates semantically correct machine code for any source program. Since certified compilers are not generally available, one is faced with writing a compiler for a restricted subset of the implementation language and certifying it to compile correctly or with adopting the second approach to the problem.

The second approach involves uncertified compilation and certified disassembly. The idea is to compile the program into machine language and then to disassemble into a readable assembly language (see Figure 4). The correspondence of the machine-language program to the assembly-language program would be assured by the certification of the disassembler; the correspondence of the assembly-language program to the original program would be established manually by whatever means are appropriate. Thus, the correspondence of the machine-language program to the high-level language program is shown by the composition of the two smaller correspondences, which together are strongly equivalent to the actual uncertified compilation.

The certified disassembly approach is predicated on the possibility of writing a certifiable disassembler. There has been some work done in this vein, including the work of C. R. Hollander [21]. Work on the certification of compilers is also proceeding; particularly interesting work in this area has been reported by L. Ragland [22] and R. L. London [23].

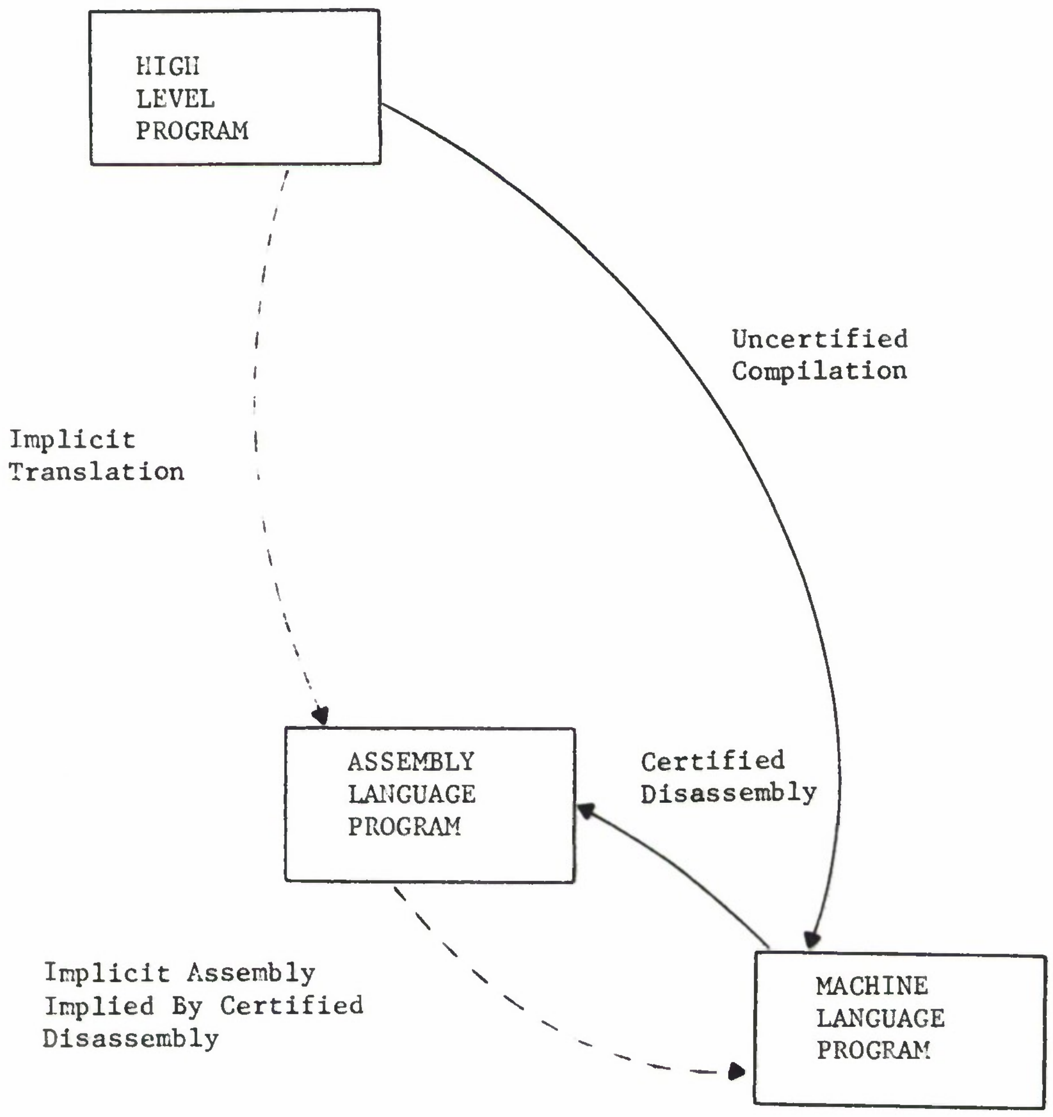

Figure 4. The Certified Disassembly Concept

RELATED WORK

The major contribution of this paper is the organization of several techniques into a cohesive software engineering framework. In this subsection, the relation of this exposition to other important areas of research will be described.

Levels of Abstraction

The validation chain we are proposing involves altering an abstract solution of a problem successively until a concrete, useable solution is reached. Despite the common term "abstract," this process is not necessarily related to Dijkstra's levels-of-abstraction [24]. In both contexts, "abstract" is used in the strict mathematical sense of "more general" and "less detailed". In our situation, the abstractness involved is that of various solutions to the same problem, solutions which vary in mode of expresssion as well as in the degree of detail involved. For Dijkstra, the degree of abstraction occurs within a single solution to a problem, the amount of elaboration present (or absent) determining the specificity of the particular description of the solution. It is eminently reasonable, when involved in the design of the Algorithmic Representation of our validation chain, to use Dijkstra's levels-of-abstraction approach, either alone or in conjunction with other techniques such as modularity.* Thus, at least in some instances, the use of levels-of-abstraction with our synthesis technique, although not required, can be beneficial.

*For a complete discussion of these techniques in the development of reliable software, see Liskov [25].

The use of levels of abstraction in the development of secure computer systems can be seen in many efforts. The MITRE work on the PDP-11/45 [9] used levels of abstraction only in the development of the algorithmic representation (just as was true for Dijkstra). The work of Neumann _et al_ [12] involves layered validation-chain components for some, and possibly all, of the relevant blocks in their methodological framework. Another approach, the development of a series of mathematical models, with each model describing a different Dijkstra-level, or virtual machine, is the approach being taken by Walter _et al_. [6]. The last approach we will mention is that envisioned by Liskov and Zilles [13] in the development of the very-high-level language CLU. Here, the model can be translated into a very abstract algorithmic representation and the top-down development of Dijkstra levels down to an implementation can be carried out within the language itself.

Program Proving Techniques

The work that has been done in the field of program proving is directly applicable to our synthesis approach. The application of these methods, however, does require care for correct use.

The vast literature of program proofs, mostly in the form of examples, can be quite illuminating in showing the difficulty of the procedure. The work of Floyd (the induction theorem) [16], Manna [18], Hoare [17], Naur (general snapshots) [26], Burstall (structural induction for recursive programs) [27], and London [23, 28] should be especially noted in this regard. It should also be mentioned, however, that the problem addressed by most of these authors involves algorithmic validity (the demonstration that a prespecified input-output relation between variables is valid) while the general problem we address involves continual state transformation. An important investigation into proving the correctness of a computer system is Lauer's thesis "Correctness in Operating Systems" [29].

Another branch of program proving that has great potential in future uses of our synthesis technique is mechanical theorem proving. The basic ideas, following a line of development from Turing through Herbrand, Davis and Putnam and Robinson, revolve around converting a program and its specifications into a theorem in symbolic logic. Correctness, against some preestablished standard, can be phrased in symbolic logic as a resolvable halting problem.* Development of automated (and eventually verified) tools to use symbolic logic in proving a program to match an input-output relation is being carried out by a number of people, including King [31], Good [32], and Igarashi _et al_. [33]. The last mentioned development provides for automatic generation of some of the internal assertions used in analysis. Further, the development of this kind of aid, based on axiomatically defined semantics as in Pascal [34], could bring substantial benefits to the proof of correspondence for the last two links in our validation chain.

There is an extensive literature in both areas of program proving. Two excellent references for examples of program proofs are the survey articles by Elspas, Levitt, Waldinger, and Waksman [35] and by London [36]. An excellent exposition of mechanical theorem proving, as well as an extensive bibliography on the subject, can be found in the Chang and Lee book [30] mentioned before.

*For a discussion and an explanation, see Section 10 of Chang and Lee's _Symbolic Logic and Mechanical Theorem Proving_ [30].

Management Techniques

The recent emphasis on management techniques such as the chief-programmer method [37] and human factors in general is in a sense orthogonal to the technique we are proposing. We are attempting to define _what_ goals should be recognized and pursued in the development of a system; management techniques address _how_ prespecified goals of an endeavor can best be achieved. Thus, when operating within a development framework such as ours, management techniques might well be used in the attainment of intermediate goals; the development of the synthesis technique itself, however, need neither presume nor endorse any particular management philosophy.

SECTION V

SUMMARY

The specific requirements of a computer security program have led us to general software synthesis technique, applicable whenever a certain degree of assurance is required of a system. The technique is an aggregate of mostly familiar techniques, combined into a cohesive software engineering discipline. The framework of the technique is the validation chain with four components representing solutions to the given problem in varying degrees of abstractness. A precise definition of correspondence has been formulated to clarify the issue of the relation between the various components of the chain. We believe the technique described herein is both practicable and useful for the synthesis of certifiable software.

REFERENCES

[1] Anderson, James P., "Computer Security Technology Planning Study," Electronic Systems Division (MCIT), AFSC, Bedford, Mass., ESD-TR-73-51, October, 1972.

[2] "ESD 73 Computer Security Development Summary," Electronic Systems Division (MCIT), AFSC, Bedford, Mass., MCI-74-1, February, 1974.

[3] Bell, E. E., and La Padula, L. J., "Secure Computer Systems: Mathematical Foundations," Electronic Systems Division (MCIT), AFSC, Bedford, Mass., ESD-TR-73-278, Vol. I, November 1973.

[4] La Padula, L. J., and Bell, D. E., "Secure Computer Systems: A Mathematical Model," Electronic Systems Division (MCIT), AFSC, Bedford, Mass., ESD-TR-73-278, Vol. II, November 1973.

[5] Bell, D. E., "Secure Computer Systems: A Refinement of the Mathematical Model," Electronic Systems Division (MCIT), AFSC, Bedford, Mass., ESD-TR-73-278, Vol. III, April 1974.

[6] Walter, K. G., *et al.*, "Primitive Models for Computer Security," Electronic Systems Division (MCIT), Bedford, Mass., ESD-TR-74-117, January, 1974.

[7] Popek, G. J., "Correctness in Access Control," *Proc.* 1973 ACM National Conference, August, 1973.

[8] Hsaio, D. K., Kerr, E. J., and McCauley, E. J., III, "A Model for Data Secure Systems (Part I)," Computer & Information Science Research Center, OSU-CISRC-TR-73-8, Ohio State University, February, 1974.

[9] Schiller, W. Lee, "Design of a Security Kernel for the PDP-11/45," Electronic Systems Division (MCIT), AFSC, Bedford, Mass., ESD-TR-73-294, December 1973.

[10] Parnas, D. L., "A Technique for Software Module Specification with Examples," CACM, Vol. 13, No. 5, May, 1972.

[11] Price, William R., "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D Thesis, Carnegie-Mellon University, June, 1973.

[12] Neumann, Peter G., *et al.*, "On the Design of a Provably Secure Operating System," presented at the International Workshop on Protection in Operating Systems, IRIA, August, 1974.

[13] Liskov, B. and Zilles, S., "Programming with Abstract Data Types," Computation Structures Group Memo 99 MIT Project Mac, March, 1974 (also presented at the Very High Level Language Symposium, Santa Monica, California, March 28-29, 1974).

[14] Karger, P. K., "On the Specification of Abstract Models of Computer Security," report for MIT course 6.891.

[15] Bell, D. E., "Correctness Can Be Categorically Affirmed," journal article in preparation.

[16] Floyd, R. W., "Assigning Meaning to Programs," Proc. Symposium on Applied Mathematics, A.M.S., Vol. 19 (1967), 19-32.

[17] Hoare, C. A. R., "Proof of a Program: FIND," Comm. ACM 14 (1971), 39-45.

[18] Manna, Z., "The Correctness of Programs," J. Computer System Science 3 (1969), 119-127.

[19] Arbib, M. A., Theories of Abstract Automata (Englewood Cliffs, 1969), p. 57.

[20] Mitchell, B., Theory of Categories (New York, 1965), p. 1 ff.

[21] Hollander, C. R., "Decompilation of Object Programs," AD-768 887, Stanford University, January, 1973.

[22] Ragland, L., "A Verified Program Verifier," Ph.D Thesis, The University of Texas at Austin, May, 1973.

[23] London, R. L., "Correctness of Two Compilers for a LISP Subset," AD-738 568, Stanford University, October, 1971.

[24] Dijkstra, E. W., "The Structure of the 'THE'-Multiprogramming System," Comm. ACM 11 (May, 1968), 341-346.

[25] Liskov, B. H., "A Design Methodology for Reliable Software Systems," Proc. FJCC, AFIPS 41 (1972), pp. 191-199.

[26] Naur, P., "Proof of Algorithms by General Snapshots," BIT 6 (1966), 310-316.

[27] Burstall, R. M., "Proving Properties of Programs by Structural Induction," The Computer Journal, 12 (1969), 41-48.

[28] London, R. L., "Proving Programs Correct: Some Techniques and Examples," BIT 10 (1970), 168-182.

[29] Lauer, H. C., "Correctness in Operating Systems," AD-753 122, Carnegie-Mellon University, September, 1972.

[30] Chang, C. L., and Lee, R. C. T., _Symbolic Logic and Mechanical Theorem Proving_ (New York, 1973), pp. 210-233.

[31] King, J. C., "A Program Verifier," Ph.D Thesis, Carnegie-Mellon University, 1969.

[32] Good, D. I., "Toward a Man-Machine System for Proving Program Correctness," Ph.D Thesis, University of Wisconsin, 1970.

[33] Igarashi, S., London, R., and Luckham, D., "Automatic Program Verification I: A Logical Basis and its Implementation," STAN-CS-73-365, Stanford University, May, 1973.

[34] Hoare, C. A. R., and Wirth, N., "An Axiomatic Definition of the Programming Language Pascal," _Acta Informatica_ 2 (1973) pp. 335-355.

[35] Elspas, B., Levitt, K. N., Waldinger, R. J., Waksman, A., "An Assessment of Techniques for Proving Program Correctness," _Computing_ Surveys 4 (1972).

[36] London, R. L., "The Current State of Proving Programs Correct," _Proc_. ACM National Conference, 1972.

[37] Mills, H., "Chief Programmer Team Principles and Procedures," IBM Federal Sys. Div., June, 1971.